Asgn1 Design

1 Introduction

Httpserver is a multi-threaded HTTP server. It responds to HTTP requests HEAD, GET, and PUT from a client. Client must send valid HTTP/1.1 requests. It uses the file resources from the directory that it is running in.

2 Data Design

I define a constant BUFFER_SIZE to be 4096 for simplicity because this is the maximum size a request header can be.

I then use a httpObject struct (which was given in the starter code) to contain relevant information for each HTTP request as follows.

- char method[5]
- char filename[28]
- char httpversion[9]
- ssize_t content_length
- int status_code
- uint8_t buffer[BUFFER_SIZE]
- int log_fd

I use a string representation to store the method, filename and httpversion. Each char array needs only be as large as the maximum possible respective string length plus one for a null-termination at the end.

I use ssize_t for content_length because I am going to be using this to compare against file sizes which are of type ssize_t.

I use int for the status_code because any valid status_code will range between 100-600 and ints are easy to work with.

I use a uint8_t buffer (byte buffer) which will be used to hold data for recv()/send() operations between client and server.

I also create a simple healthObject struct used to contain two integers representing the number of errors and entries a server has encountered.

I use a worker struct to contain relevant information for each worker thread as follows

- int id
- int client_sockd
- int log_fd
- pthread_t worker_id
- struct healthObject* p_health

- pthread_cond_t condition_var
- pthread_mutex_t* lock
- bool available

id contains an integer that is used to refer to the worker

3 Component Design

The purpose of main will be to check the usage, initialize the server, and handle HTTP requests.

3.1 Main

The job of main will be to check the usage, initialize the server, create and initialize worker threads and then go into a loop to accept new connections. Main will also allocate space for a healthObject and create a pointer to it that will be used by handle_task to track the number of errors and entries.

3.1.2 Accepting connections

To accept new connections and assign them to a worker thread, I used a simple loop that will continuously loop through the array of workers round-robin style, checking if the worker is available, until an available worker is found. When an available worker is found, it will give that worker the accepted client socket, signal the worker to start working.

3.1.3 Handle_task

Because p_thread create requires a single function that takes and returns a void pointer, I create a function ‘handle_task’ that will deal with each http request. Handle_task takes a pointer to a worker thread as an argument. It will first create an httpObject, then enter a while(true) loop, wait to be signaled by the dispatcher thread, then read, process and construct a response given a client socket, increment the entries/errors, write to the log if logging is enabled, and mark the worker as available.

3.2 Initialize

The starter code was written to initialize a server. I can’t speak much as to how it is implemented.

3.2.0 Multi-threading

To ensure that seperate threads do not step on each other, mutexes must be implemented to lock critical sections of the code. I have identified three critical areas where potential errors may occur: process_http_request(), construct_http_response(), and write_log(). Process and request present a possible hazard because two subsequent requests could attempt to modify the same file at the same time. Additionally, write_log() must be given to one thread at a time because it will always write to the same log file. Reading a request is safe because each thread has its own httpObject struct that it can read to.

I believe it would be possible to improve this by only locking the file that a thread is currently working on. This would ensure that two threads could process requests to separate files. I also believe I could change how write_log() is implemented so that it takes an offset that is equal to how many bytes are or will be present in the log file. Then each thread only writes to the space in the log file that it has been given.

3.3 Logging

To implement logging, I wrote a helper function write_log() that takes a httpObject struct. Using the information in the httpObject, it will first determine if it was a valid request and write the first line, then determine if file data was successfully transferred by checking the status code and method. If so, it will encode and write the transferred file data to the log file specified by the log_fd present in the httpObject.

3.2.1 Handle HTTP requests

I broke this part down into three separate functions: read_http_request, process_request, and construct_http_response.

Read_http_request uses recv() to read data from a client and stores it in the struct httpObject buffer. I then null terminate it, and use strtok() to parse out the method, filename, and HTTP version which should all be consecutive in the request, and store them in their respective fields in the httpObject. I then use strstr() to find the substring “Content-Length:” if it exists and store that as well.

In process_request, I am primarily error checking, setting status codes and content length. I perform error checking using the httpObject to ensure that the HTTP version is valid, and that the filename is valid. If so, I use stat() to extract information about the specified file into a stat struct. I then check the httpObject method.

For a GET or HEAD method, I first check if the file exists using the return value of stat(). Then I check the file permissions for the user read bit by performing a bitwise AND operation with S_IRUSR and checking the outcome. If either of these fail, I set the status_code accordingly and set content_length to 0. Otherwise, I set the status_code to 200, and the content_length to the size retrieved from stat(). I do not send file data for a GET request in process

For a PUT method, I similarly check that the file exists and the user write bit is set. I then open the file specified by the httpObject filename and use strstr() on the httpObject buffer to determine if there is anything after the double carriage return line feed that marks the end of a http request header. If so, I write to the opened file the contents of httpObject buffer starting from the end of the double crlf, keeping track of the total bytes written, and ensuring it is less than the specified content length. After this, I use a while loop to recv() and write() until the total bytes that have been written is greater than or equal to the httpObject content length.

For any other method, I simply set the status code to 400.

In Construct_http_response, I simply use snprintf and the fields of the struct httpObject to construct a string with the http version, status code, status message, and content length. I then send that string to the client.

I then check if the method is GET and status code is 200 to determine if I have to send file data. If so, I open the file and send the file data using a while loop that checks that the total bytes sent is less than httpObject content length.